

Lecture 4

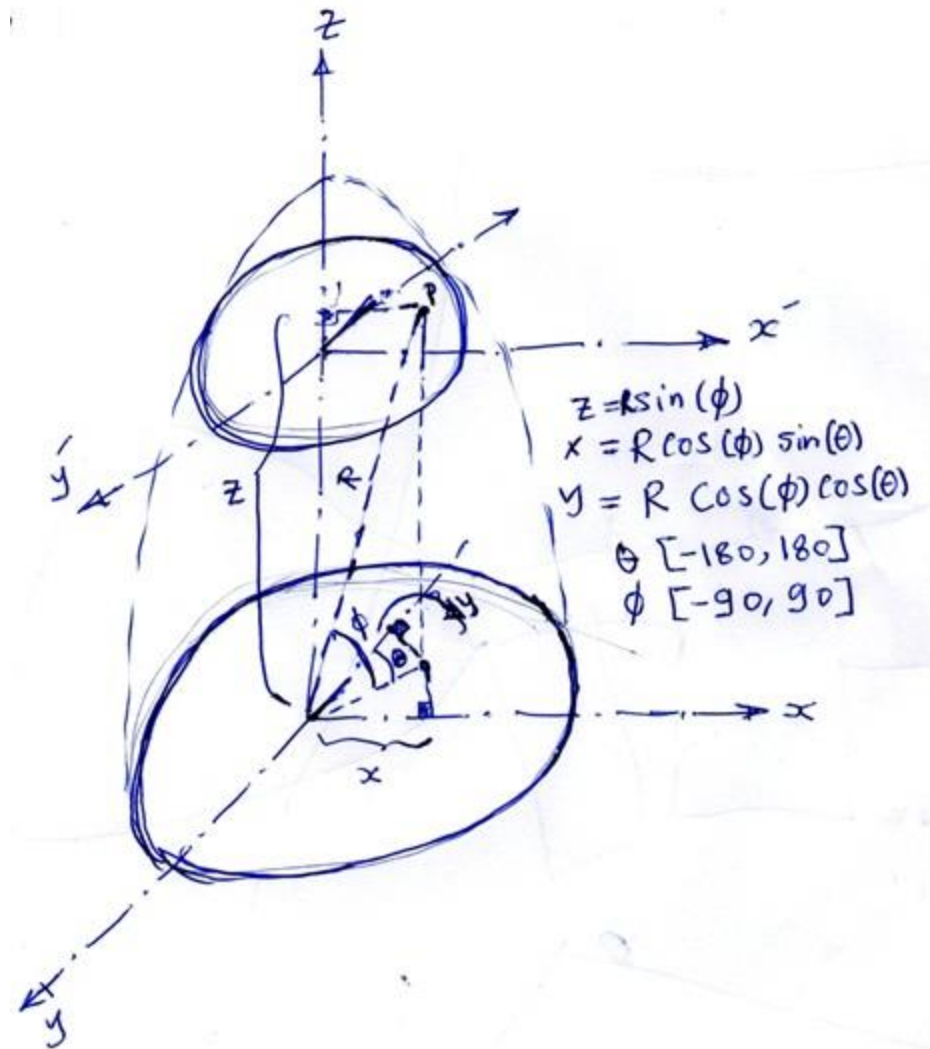
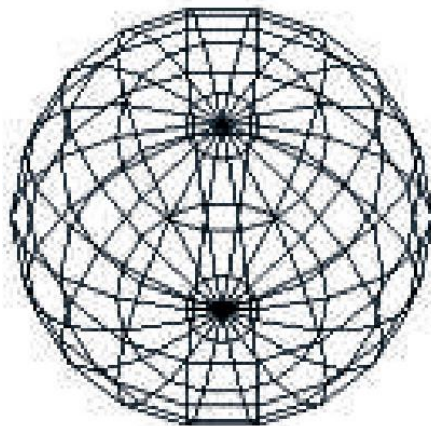
Object approximation

Text, Curved objects, Attributes, Colors

Approximating objects using polygons

A sphere can be approximated by triangles at the two poles and quadrilaterals in-between

$$\begin{aligned}x(\theta, \phi) &= \sin \theta \cos \phi, \\y(\theta, \phi) &= \cos \theta \cos \phi, \\z(\theta, \phi) &= \sin \phi.\end{aligned}$$



First: Sphere triangles at poles

Sphere approximation

Second: Sphere quadrilaterals

```
for(phi=-80.0; phi<=80.0; phi+=20.0)
{
    phir=c*phi;
    phir20=c*(phi+20);
    glBegin(GL_QUAD_STRIP);
    for(theta=-180.0; theta<=180.0; theta+=20.0)
    {
        thetar=c*theta;
        x=sin(thetar)*cos(phir);
        y=cos(thetar)*cos(phir);
        z=sin(phir);
        glVertex3d(x,y,z);
        x=sin(thetar)*cos(phir20);
        y=cos(thetar)*cos(phir20);
        z=sin(phir20);
        glVertex3d(x,y,z);
    }
    glEnd();
}
```

```
glBegin(GL_TRIANGLE_FAN);
    glVertex3d(0.0, 0.0 , 1.0);
    c=M_PI/180.0;
    c80=c*80.0;

    z=sin(c80);
    for(thet=-180.0; theta<=180.0; theta+=20.0)
    {
        thetar=c*theta;
        x=sin(thetar)*cos(c80);
        y=cos(thetar)*cos(c80);
        glVertex3d(x,y,z);
    }
    glEnd();

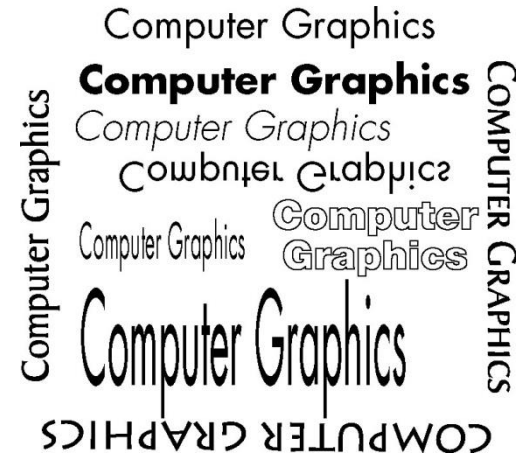
glBegin(GL_TRIANGLE_FAN);
    glVertex3d(0.0, 0.0, -1.0);
    z=-sin(c80);
    for(theta=-180.0; theta<=180.0; theta+=20.0)
    {
        thetar=c*theta;
        x=sin(thetar)*cos(c80);
        y=cos(thetar)*cos(c80);
        glVertex3d(x,y,z);
    }
    glEnd();
```

Text in OpenGL

Raster text is simple and fast. Characters are drawn as rectangles of bits called bit blocks. Each block defines a single character by the pattern of 0 and 1 bits in the block.



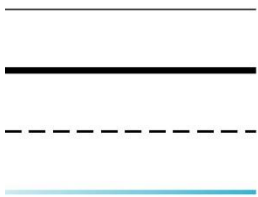
Stroke text is constructed as are other geometric objects. We use vertices to define line segments or curves that outline each character. If the characters are defined by closed boundaries, we can fill them. The advantage of stroke text is that it can be manipulated by our standard transformations and viewed like any other geometrical objects.



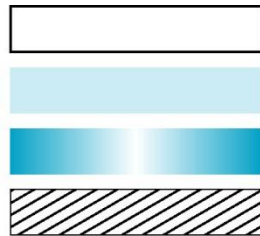
```
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, c)
```

OpenGL state: Attributes

Some state attributes that affect lines (a) and polygons (b)



(a)



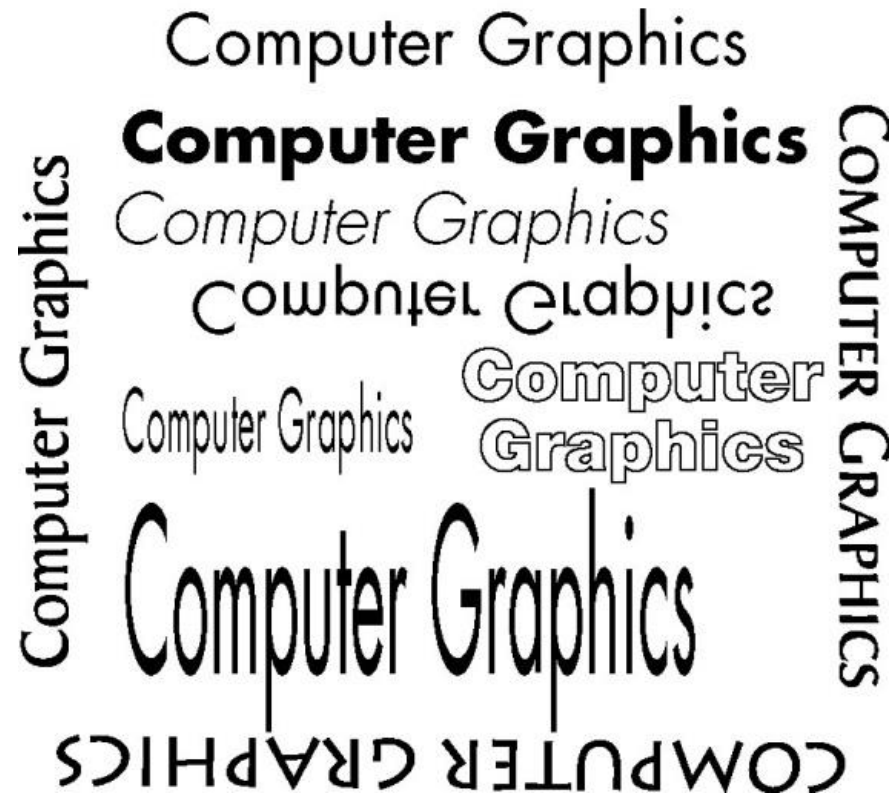
(b)

Attributes may be associated with primitives at various points in the modeling and rendering pipeline. OpenGL employs immediate-mode graphics(primitives are not stored in the system but rather passed through the system for possible rendering as soon as they are defined. The current values of attributes are part of the state of the graphics system.

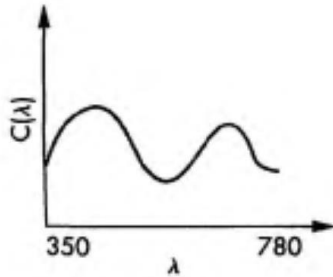
Example: line color is a state attribute rather than a line attribute (contrast this with OOP)

OpenGL state: Attributes

In systems that support stroke text as primitives, there is a variety of attributes. The effects of some of these is shown in the figure



The figure displays the text "Computer Graphics" rendered in multiple styles to illustrate different OpenGL attributes. The styles include: a standard sans-serif font; a bold sans-serif font; an italicized sans-serif font; a serif font; a monospaced font; a font with a thick stroke; a font with a thin stroke; a font with a dotted outline; a font with a solid outline; a font with a shadow; a font with a gradient; a font with a 3D effect; a font with a wavy or distorted appearance; and a font with a different color (though all are black in this image). The text is arranged in a circular pattern, with some instances rotated 90 degrees clockwise and others 90 degrees counter-clockwise.

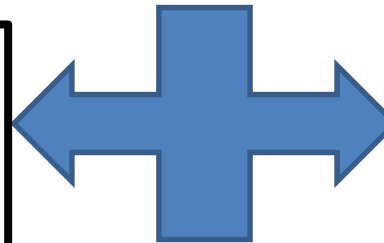


Physical light: Visible range

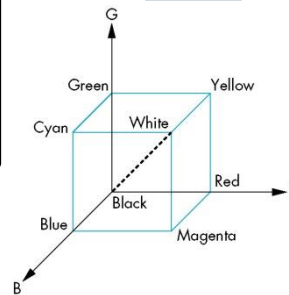


Humans can not perceive the entire distribution for a given color because they have only three types of color preceptors each perceive a specific range(color). Then, any color can be modeled to human vision as three color components: Red, Green and Blue. The freq ranges of these components are selected so that each components causes a different preceptor type to react

Additive color model RGB: Using three wavelength rages (colors) generators (close enough to each other to be perceived as a single averaged color). The intensity (amplitude) of each components determine to what extend this component participates in the production of the average color. Example CRT, video projector



Subtractive color model: Start with a white sheet. If you need Red at a specific point, put pigment there that absorb all the incident light except the Red (Yellow and Magenta pigments). If you need a white point, put no pigments(Cyan, Magenta, Yellow)= (0,0,0). If you need a black points, put pigments (Cyan, Magenta, Yellow)= (1,1,1). Used in printing



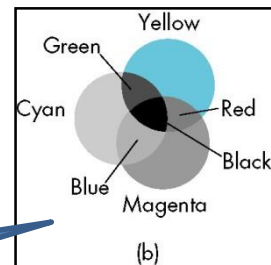
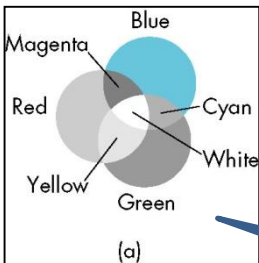
Display buffer: RGB



Color printing buffer
 $CMY=(1-R,1-G,1-B)$

Black

White



Specifying colors attributes in OpenGL

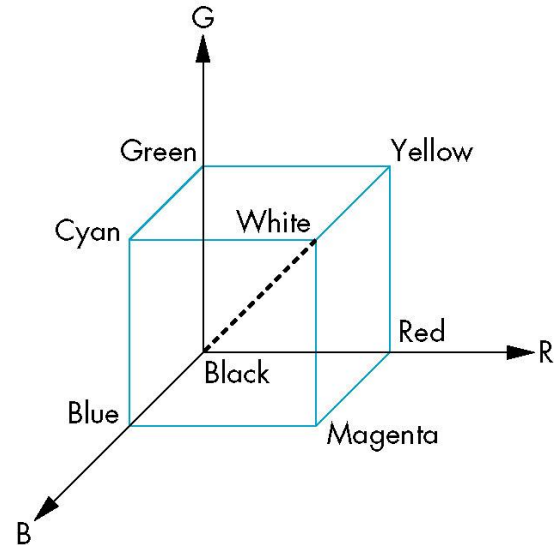
RGB color model is used in a device independent way. **Cancel Indexed color reading**

Each pixel is represented by three color buffer. The width of each of the three buffer is the same on the same device but may be different on another.

To specify the color in a device independent way, OpenGL use the color cube of 1 as a side length

This means that the color primaries are normalized (0 to 1) and the actual three buffer are filled using scaling

For example, 1 is filled as 255 in 24 bit color system that uses 8 bit for each color



```
glColor3f(1.0, 0.0, 0.0);
```

Setting the current drawing color to RED

The drawing is continued in RED until changed

```
glClearColor(1.0, 1.0, 1.0, 1.0);
```

Set the clear color: Opaque white, The glClear() is used to clear the drawing window using the clear color

Sometimes, we specify the color using four components RGBA. The fourth parameter (Alpha) is used in fog/transparency/Opacity modeling

Device dependent attributes in OpenGL

```
void glPointSize(GLfloat size);
```

Sets the width in pixels for rendered points; *size* must be greater than 0.0 and by default is 1.0.

```
void glLineWidth(GLfloat width);
```

Sets the width, in pixels, for rendered lines; *width* must be greater than 0.0 and by default is 1.0.

Version 3.1 does not support values greater than 1.0, and will generate a `GL_INVALID_VALUE` error if a value greater than 1.0 is specified.

Read further

OpenGL[®]
Programming Guide
Seventh Edition

*The Official Guide to
Learning OpenGL[®], Versions 3.0 and 3.1*

*Dave Shreiner
The Khronos OpenGL ARB Working Group*

The line width or pixel size may appear differently on different devices because they may have different pixel dimensions

Report discussion

Previous lecture report:
Problem 2.3

Next lecture report:
Problem 2.15

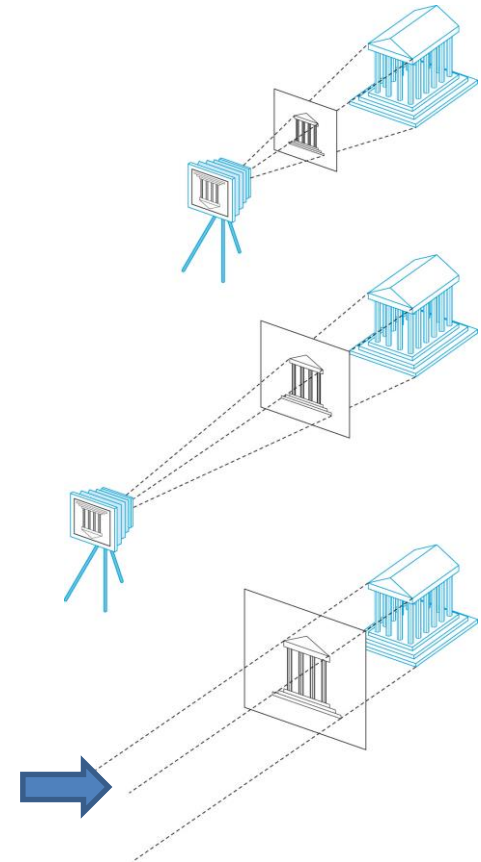
In OpenGL, we can associate a color with each vertex. If the endpoints of a line segment have different colors assigned to them, OpenGL will interpolate between the colors as it renders the line segment. It will do the same for polygons. Use this property to display the Maxwell triangle: an equilateral triangle whose vertices are red, green, and blue.

Viewing

We now can specify variety of objects in the world using the world coordinates. Which of these objects will appear on the screen window? How they appear? This depends on where the camera or viewer is located in the world? How it's or he is oriented? How is the projection done? All this is determined by specifying the viewing.

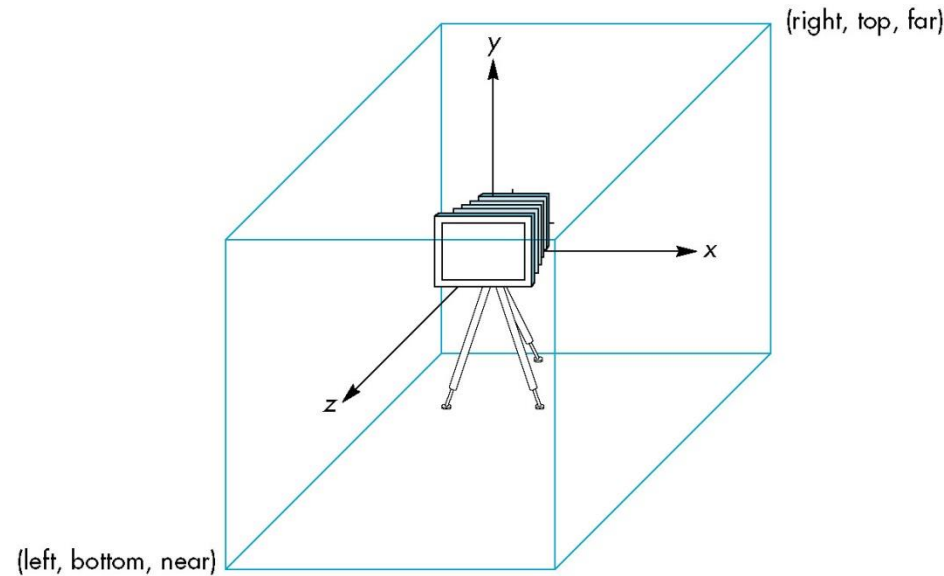
Default viewing in OpenGL

- Orthographic view
- Infinitively long lens (parallel projection)
- No matter how the camera is far from the objects the image is the same
- The center of projection is replaced by direction of projection parallel to the positive Z axis
- Projection plane at $z=0$ and is parallel to the xy plane
- The reference point by which the viewing volume is specified in the origin



Viewing: Default in OpenGL

- Projection lines are perpendicular to the $z=0$ plane and are parallel to the positive z axis or negative z axis depending on the location of objects w.r.t. to the projection plane
- Objects that are behind the viewer or the camera appear as long as they are in the viewing right-parallelpiped volume. Hence sliding the projection plane inside the viewing volume does not affect the resulting image (you can consider that the scene is projected on the near plane)



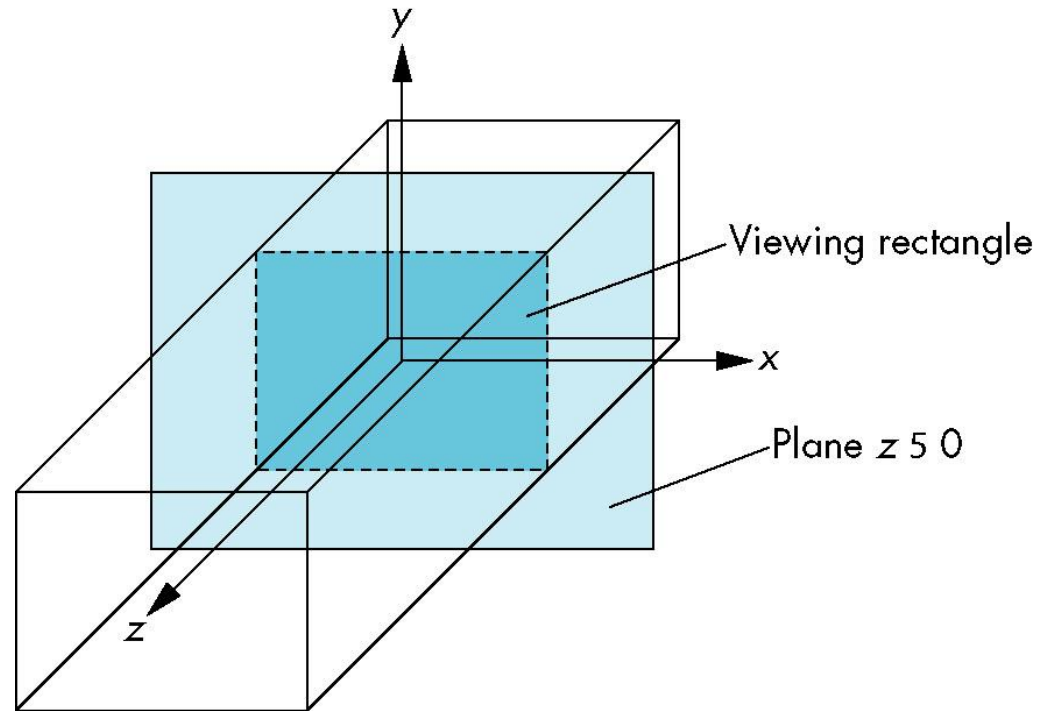
Viewing volume

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
             GLdouble top, GLdouble near, GLdouble far)
```

All the parameters are distances measured from the camera reference point (default to the origin in the world coordinates)

Viewing: Default in OpenGL

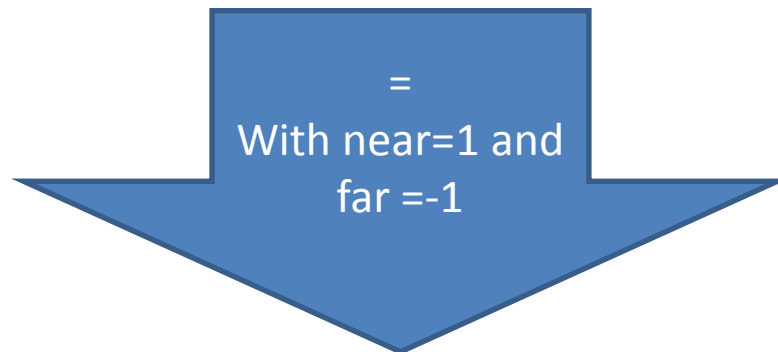
Two dimension drawing with Orthographic view: Just specify objects in any plane parallel to the projection plane (including the projection plane itself)



- Any object (or part of object) inside the viewing volume will be projected on the $z=0$ plane and appear in the image no matter it's before or after the $z=0$ plane (if the depth is enabled, you can consider that the scene is projected on the near plane)
- If the viewing volume is not specified, OpenGL defaults to 2x2x2 cube with the origin at the center

Two-dimensions viewing

```
void gluOrtho2D(GLdouble left, GLdouble right,  
                GLdouble bottom, GLdouble top)
```



```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,  
             GLdouble top, GLdouble near, GLdouble far)
```

Viewing and matrix mode

Mathematically, inside the pipeline, The projection is done by multiplying the vertices (in camera coordinates) in a Projection matrix . Setting the viewing using `glOrtho` or `gluOrtho2D` changes the contents of this matrix. It's also can be specified by loading a matrix directly. The projection matrix is a state machine variable. It's changed in accumulation manner: the new setting is calculated onto the existing setting internally by matrix multiplication. Initially, the projection matrix is initialized by the identity matrix

Another state matrix is the Mode-View matrix. It controls transformation of objects specified in the world such as (translation, rotation,...). It's changed using transformation function calls or by loading a matrix directly. As the projection matrix, it's changed in an accumulation. And is reset by loading the identity matrix

Viewing and matrix mode

- It's a single set of function that manipulate the two matrices. Therefore, the matrix must be correctly chosen before changing it using the function. The matrix choice is determined by another state variable changed by the function: `glMatrixMode()`
- A good practice is to make a specific mode as the default. This reduces the risk of changing a matrix while you intent to change the other in a complex program

Hence, the preferred way to set the projection

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
gluOrtho2D(0.0, 50.0, 0.0, 50.0);  
glMatrixMode(GL_MODELVIEW);
```